Parveen Khurana

124 Followers About Following

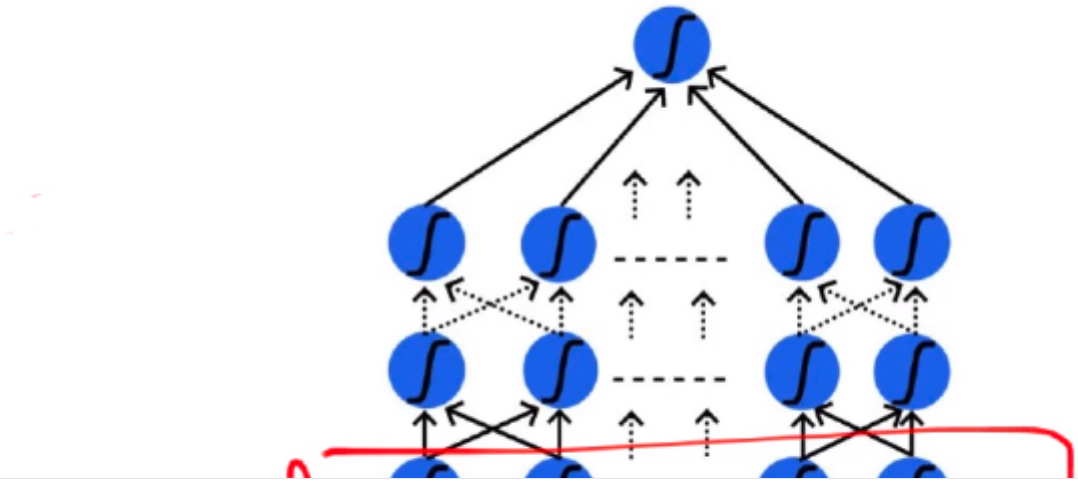
Convolutional Neural Networks — CNN

Parveen Khurana Feb 15, 2020 · 11 min read

In the previous article, we discussed the Convolution Operation. In this article, we discuss how the convolutional operation or in general the Convolutional Neural Network relates to Neural Networks and Deep Learning in general. This article covers the content discussed in the Convolutional Neural Networks module of the Deep Learning course and all the images are taken from the same module.

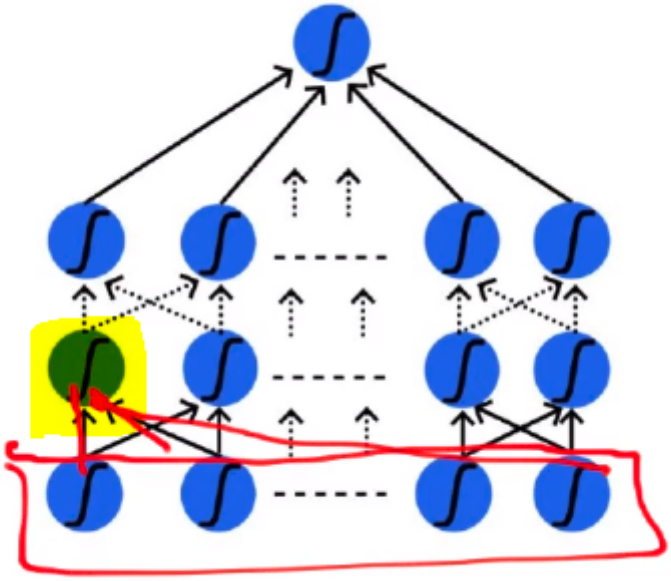
The relation between the convolutional operation and the neural networks:

Let say we are given an input image of size **'30 X 30 X 3'**(i.e 2700 pixels) and our **task is to predict if this image contains a signboard or not**(Binary Classification problem), if we unfold these 2700 values into one single vector then this vector would act as the input layer in the fully connected neural network depicted below where each neuron in this input layer is connected to all the neurons in the next layer:



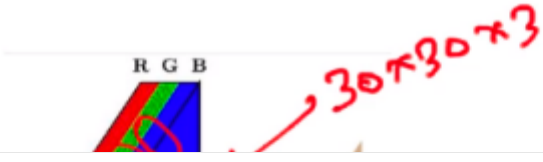
After the input layer, we have a few hidden layers containing some neurons say 100 neurons each. And then we have the output layer containing only one neuron which would be a sigmoid neuron.

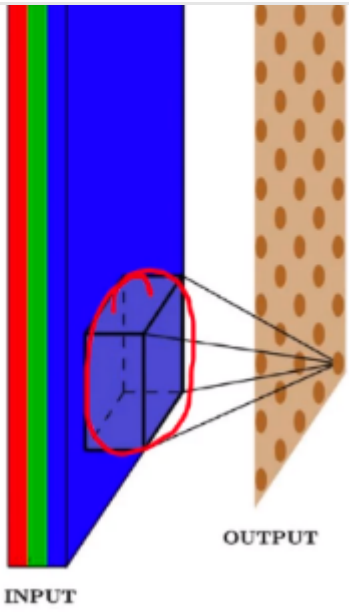
For each of the 2700 neurons(input values), we are multiplying these values with certain weights and computing the pre-activation value of the neuron highlighted in the below image:



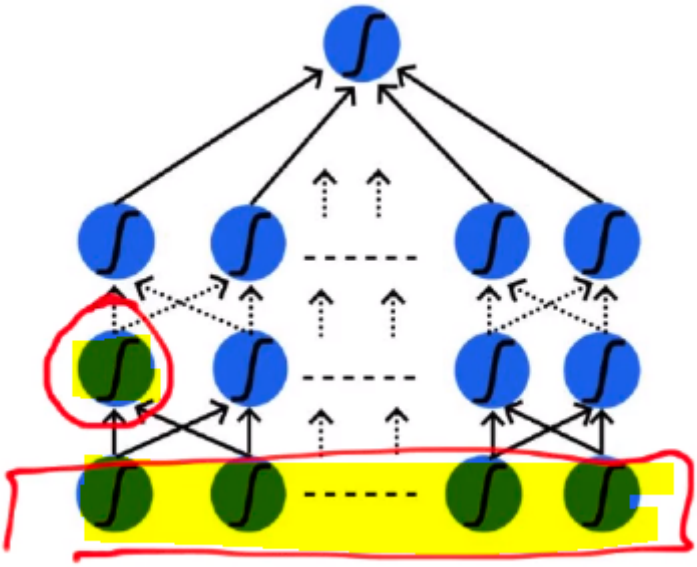
And we are doing this multiple times i.e for all the neurons in the first intermediate layer(different weights would be associated with different inputs for computing the pre-activation of different neurons in the first intermediate layer).

And now if see the convolutional operation, we find that there also we are doing the same thing i.e we have the inputs(although arranged in a different manner compared to fully connected neural networks), we are taking the weighted average of those inputs(kernel) and we get a re-estimated value of the inputs:

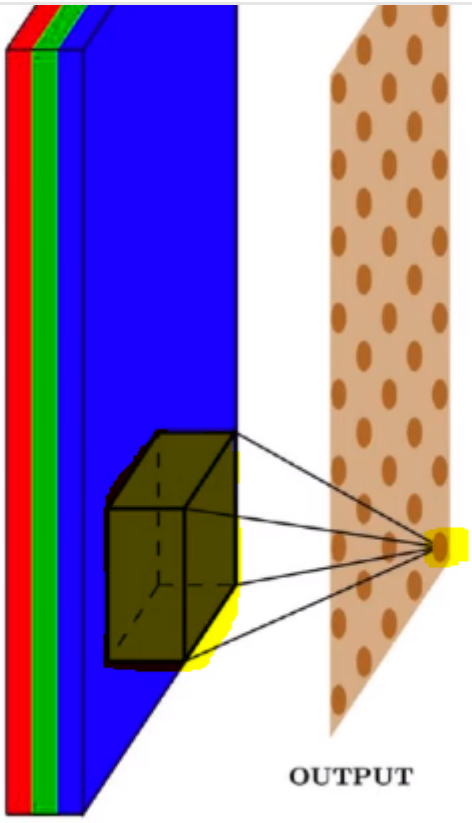




In the case of **Fully Connected Neural Networks**(FCNN), **to compute the value of one neuron in the hidden layer, we take the weighted average of all the neurons from the previous layer** whereas **in the case of CNN's, we take the weighted average of a neighborhood of values/inputs and not all the inputs and the weights are contained in the kernel.**

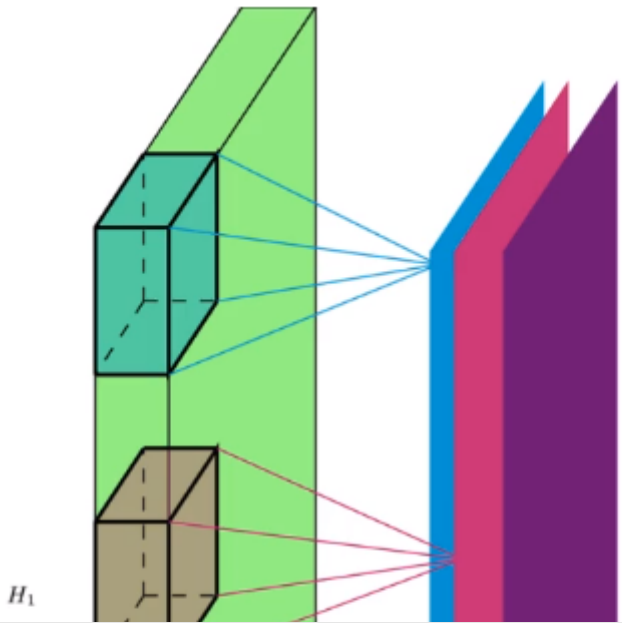


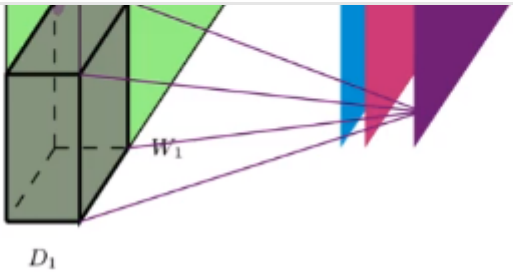
Fully Connected Neural Network

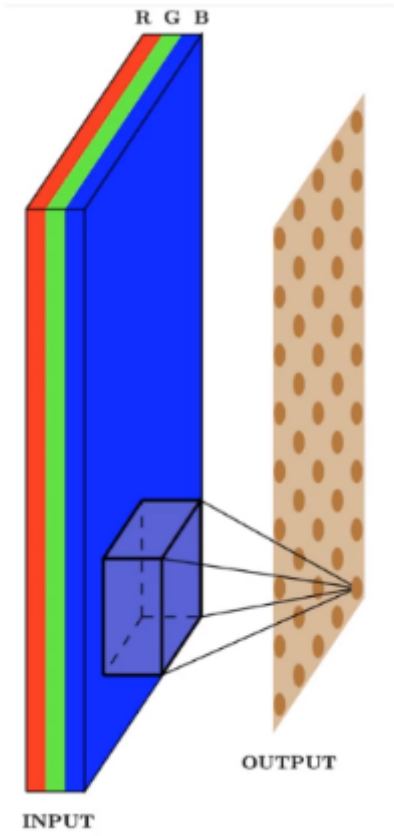


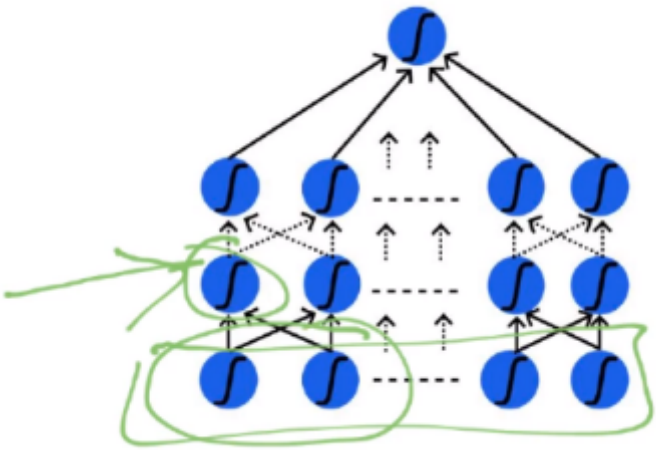
Convolutional Neural Network

So, the only difference is that in the case of FCNN, we consider all the inputs to compute the value of any of the neurons whereas, in the case of CNN, we consider only a neighbor of the inputs(we can consider this situation as that the weights of the other inputs are 0). Further, in CNN, we compute multiple such outputs at one go(we can think of it as computing multiple outputs at the same hidden layer using a different set of weights/kernels).

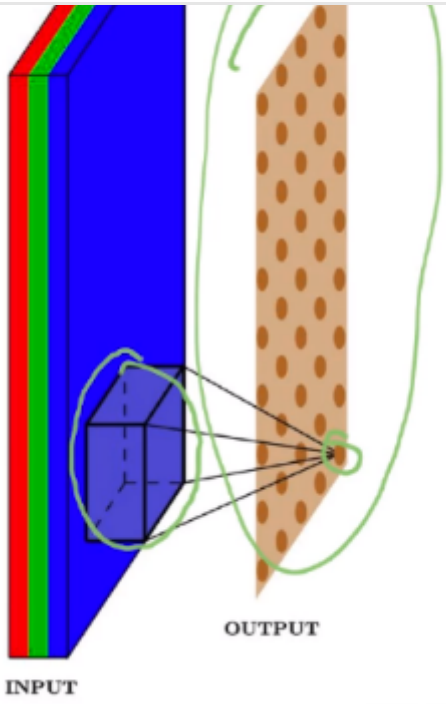








All the neurons in the input layer are used to compute the first output value in the next layer



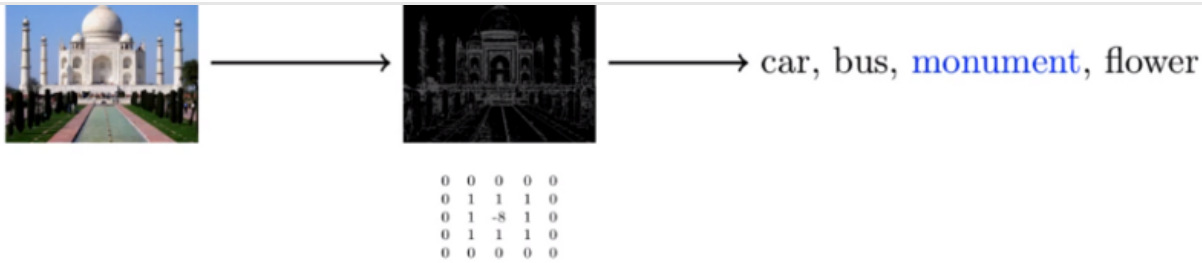
In CNN, only a portion of the input is used to compute a particular value in the output layer

In essence, we can say that in fully connected neural networks the convolutional operation is used as we take the weighted sum of the all the neurons and the same thing is there in CNN as well with the only difference that we consider only a small neighborhood to compute the weighted sum instead of taking all the neurons/pixels/values into consideration.

*Why not let the network learn the **feature representation** also ?*

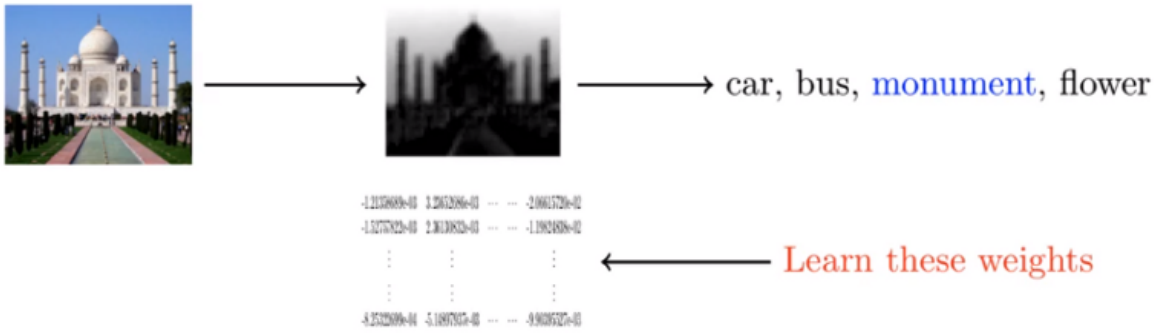
Instead of passing all the input pixels directly to the classification layer, we try to find better representations of the input, say the first two hidden layers represent two different representations of the input image where we have a fewer number of neurons as compared to the inputs in the layer. We try to train the weights of these representation layers in a way that the overall loss is minimized.

In Machine Learning, we try to have a better representation of the input by using static feature engineering for example, in the below image, we pass the input image through the edge detector and then pass this representation to the classifier and learn the weights of the classifier. So, in a way, the weights of this edge detector are fixed already or to say are static or fixed.



The question to ask here is who we are to decide that the edge detector is the right set of weights to use here, the edge detector has a certain configuration. It might be the case that in certain cases, blur detector or some other detector is the best way to get the better representation of the image which we can than pass to the neural network classifier layer.

We can also learn the weights of this representation layer instead of using the static weights or any fixed weights

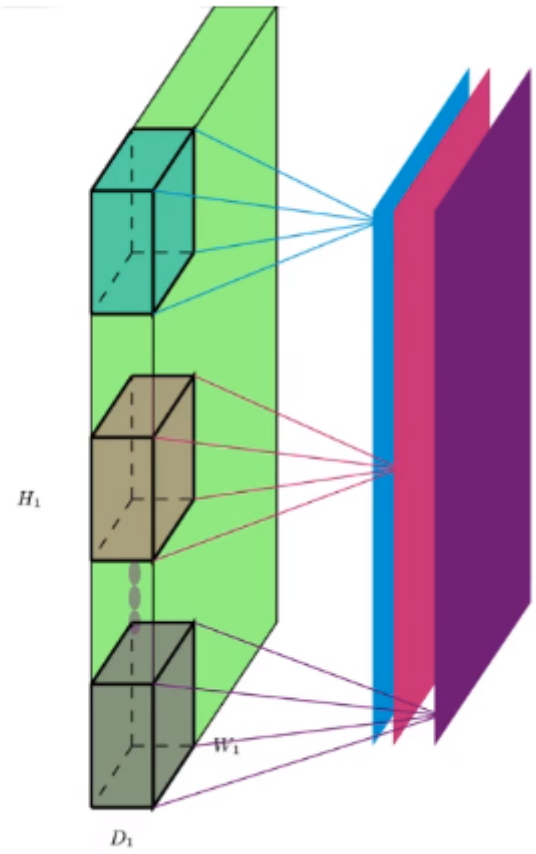


We will learn these representation layers weights using backpropagation and our main goal would be to train these weights in a way that the overall loss is minimized.

The next question is why only one feature representation what if in some case this edge detector is not important and blur detector is important or maybe some other detector is important. So, how do we know which filter and how many filters are required?

So, instead of learning one transformation of the inputs why not learn multiple transformations of the input.

We take an input, we apply multiple filters to it, each filter will give us one hidden representation and we can concatenate all of these into one big vector and we pass this representation to the final output layer. We train our network with the objective that the weights of all the filters be such that the overall loss is minimized.

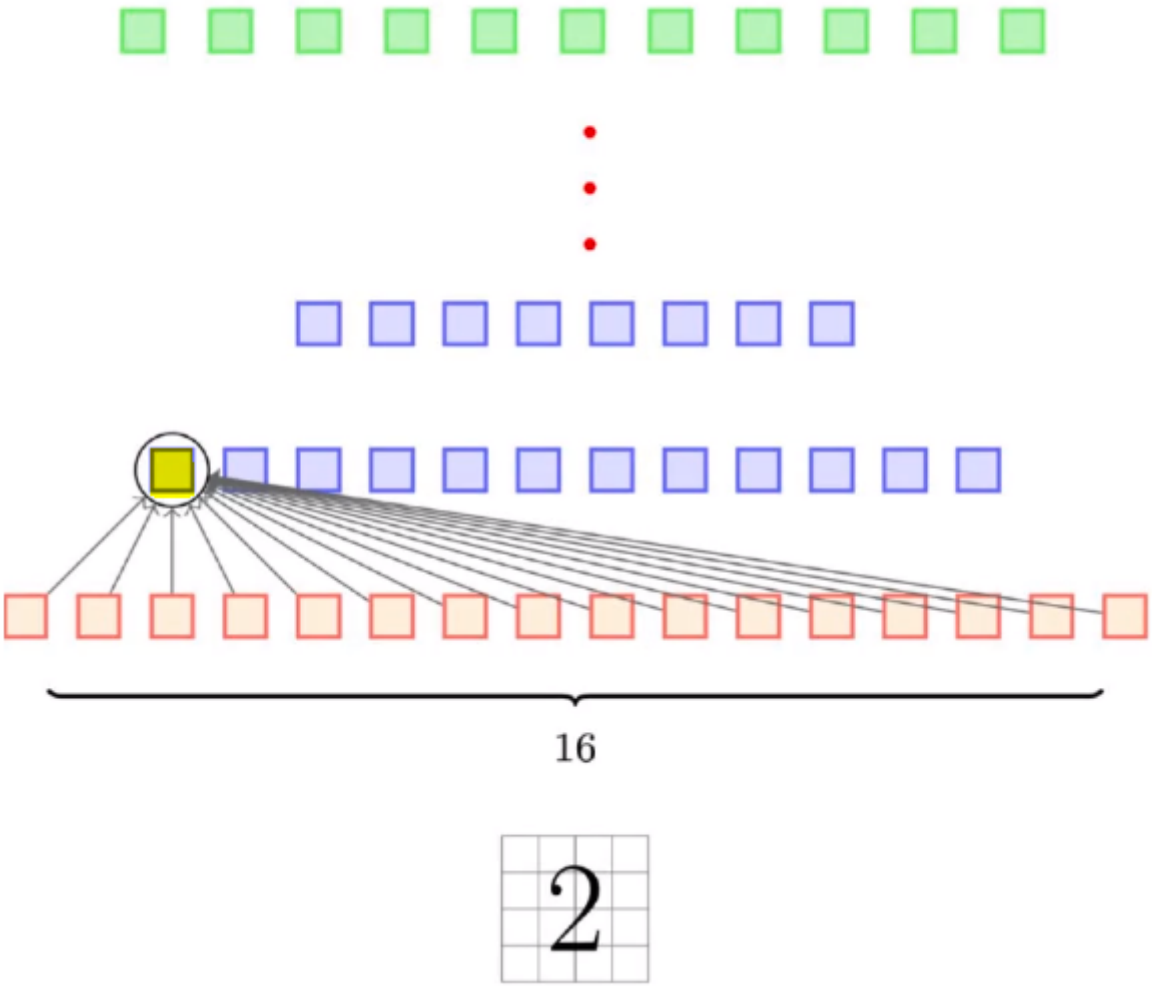


So, instead of learning a single feature representation, we learn multiple layers of feature representations.

We can have multiple intermediate layers each corresponding to the feature representations. So, now the key idea here is instead of ourselves deciding the best filter to transform the image, we are learning the weights of the feature representation as well using backpropagation.

Sparse Connectivity and Weight sharing:

Let's say our goal is to build a classifier to identify one of 10 digits from the input image. And our input is of size '**4X4**' i.e we have 16 pixels in total:



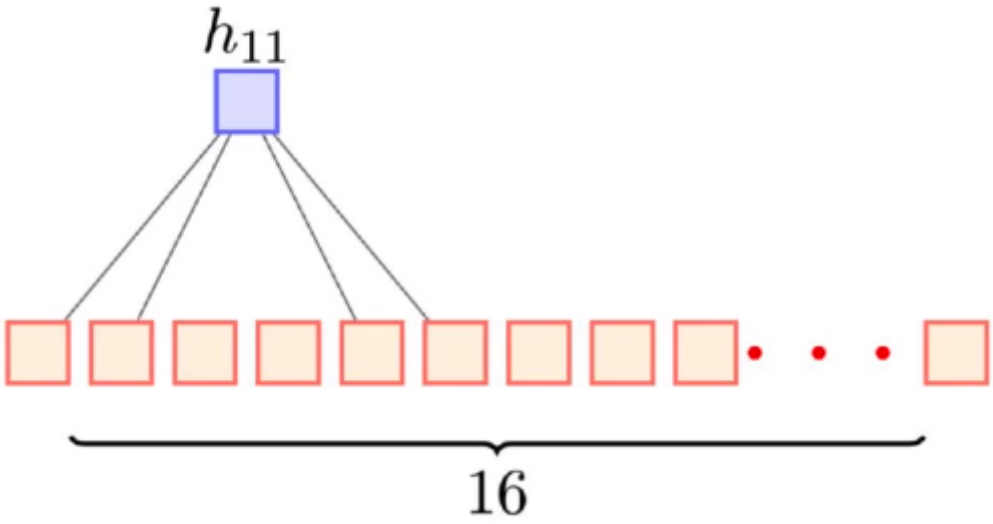
fully connected network for image classification

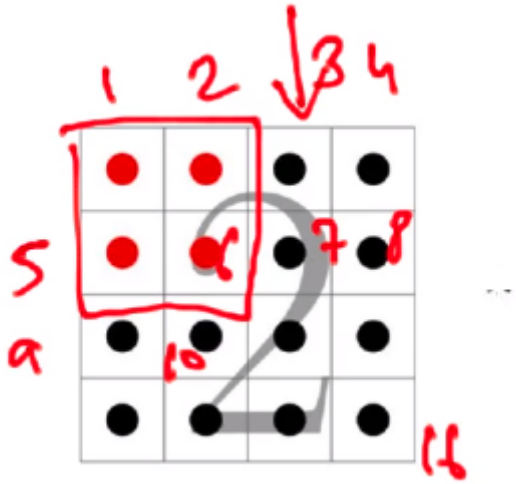
If we look at the highlighted neuron in the above image, it is clear that it is computed taking a weighted sum of all the neurons(red rectangles) from the previous layer. Similarly, any neuron in any of the layers takes a weighted sum of all the neurons from the previous layer in case of a fully connected neural network.

We pass this input through multiple layers and finally, we have 10 neurons/output(corresponding to the 10 digits from 0–9) in the final output layer.

In the case of CNN's, if we use the kernel of size **'2X2'** and place it over **'2X2'** portion of the input image(yellow highlighted), we can compute the value of the one single neuron in the intermediate layer using these 4 pixels from the input image.

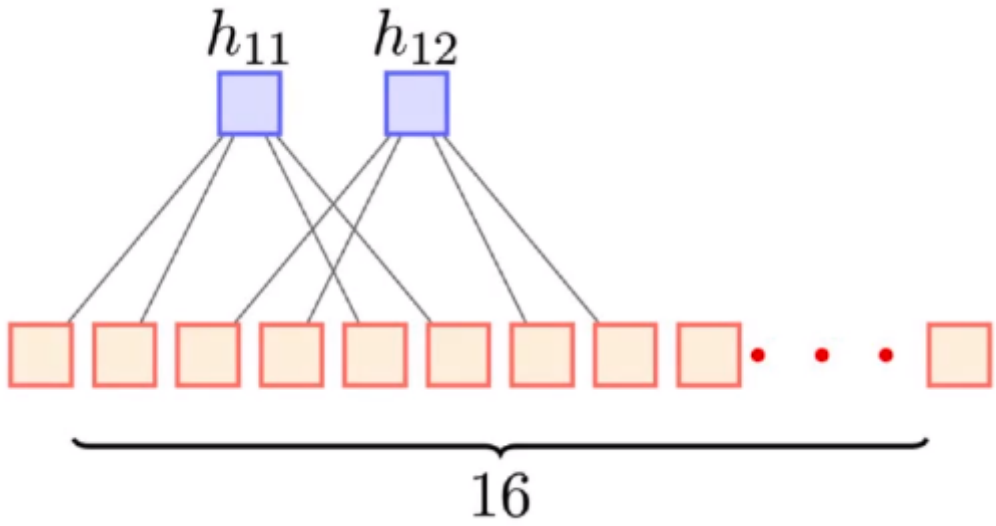
If we flatten out this **'2X2'** image, the below image shows the weights contributing(we number the weights starting from 1 all the way up to 16 as in below image) to the computation of **'h11'**





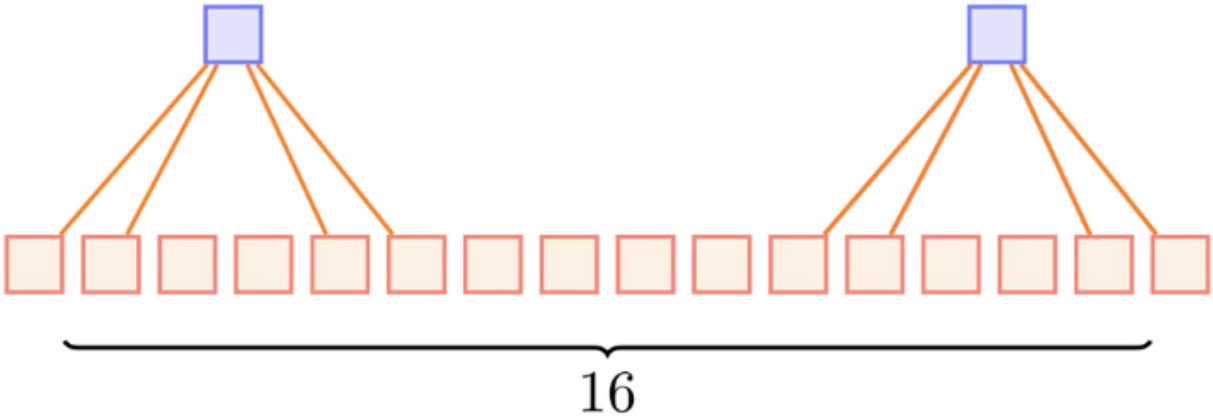
So, **'h11'**(11 is for the first neuron in first intermediate layer) is computed using 4 values(in red rectangle in above image) instead of taking all the 16 values into consideration or the other way to look this is that all the 16 values are taken into

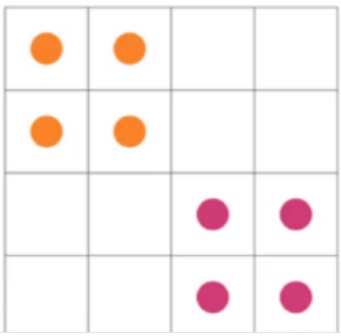
Similarly, for the computation of second neuron in the intermediate layer, we again consider only 4 weights:



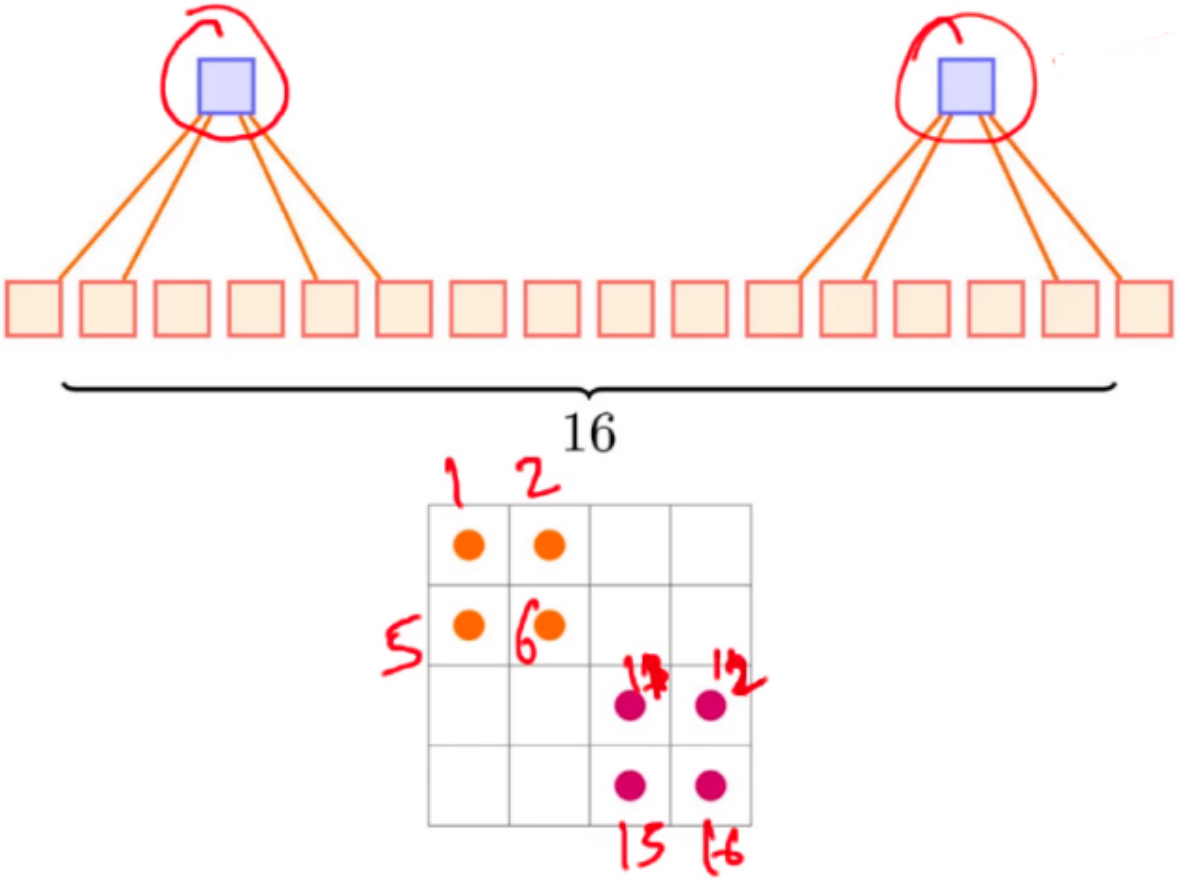
This property of CNN is termed as **sparse connectivity** i.e most of the connections have 0 weights.

The second property which makes CNN's different from fully connected neural networks is that CNN's use **weight sharing**:

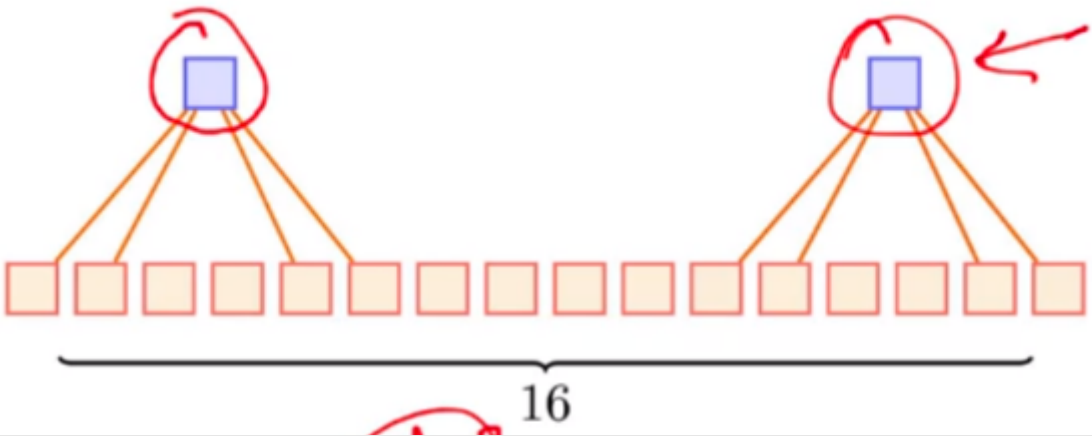


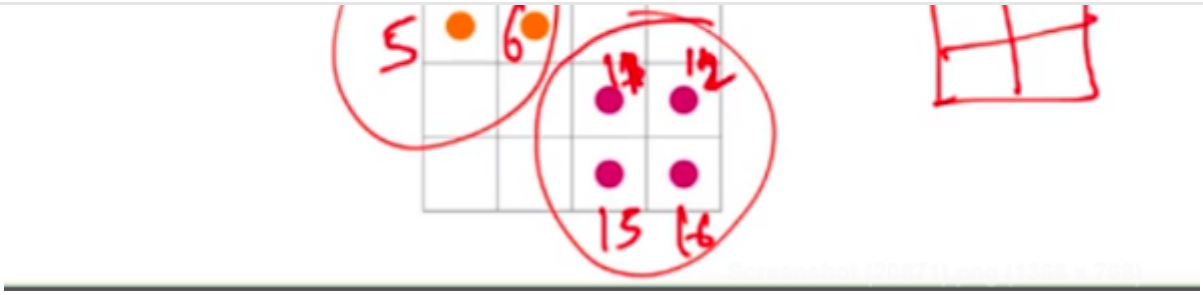


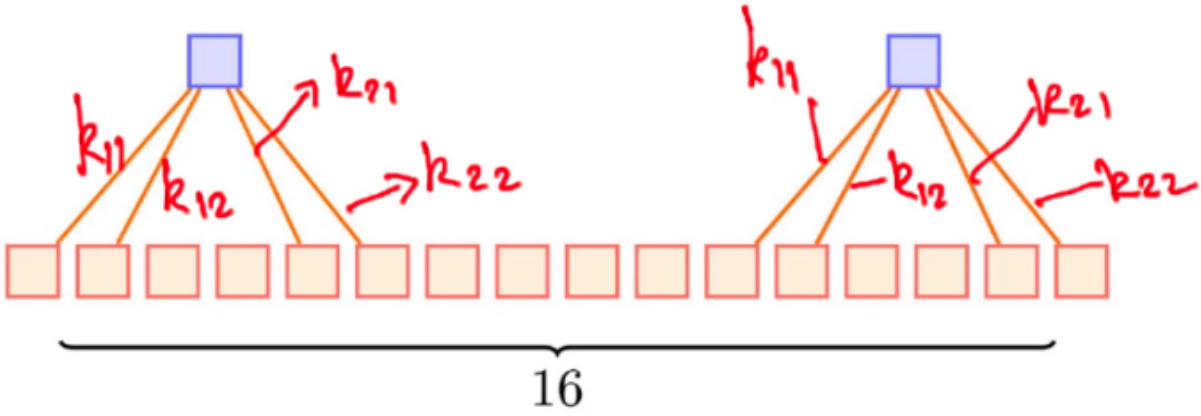
corresponding output neuron to explain this property of CNN's. Let's number the pixels from 1 to 16, then we have

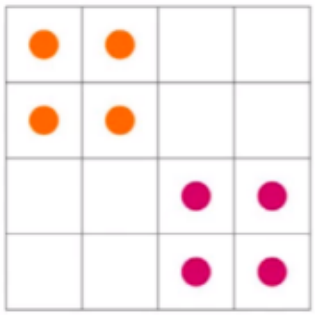


Now to compute the **'h11'**(11 denotes the first neuron in the first output layer), the pixels number 1, 2, 5, 6 are used and to compute the second output(shown in the above image), pixel number 11, 12, 15, 16 are used. The same **'2X2'** kernel is used to compute these outputs(the same kernel is sliding horizontally and vertically).

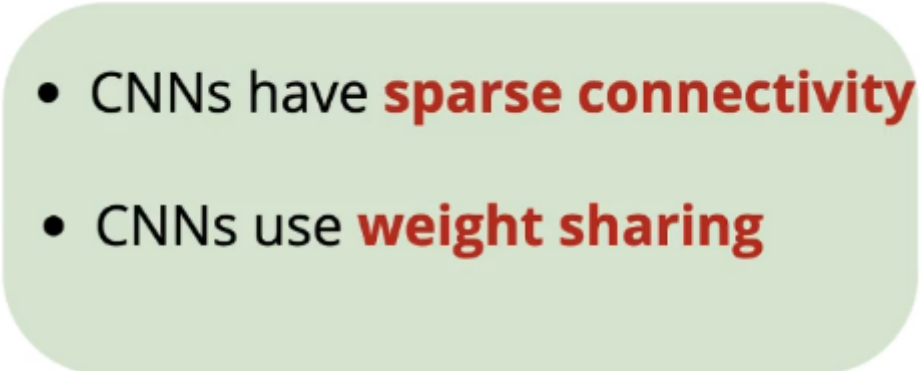






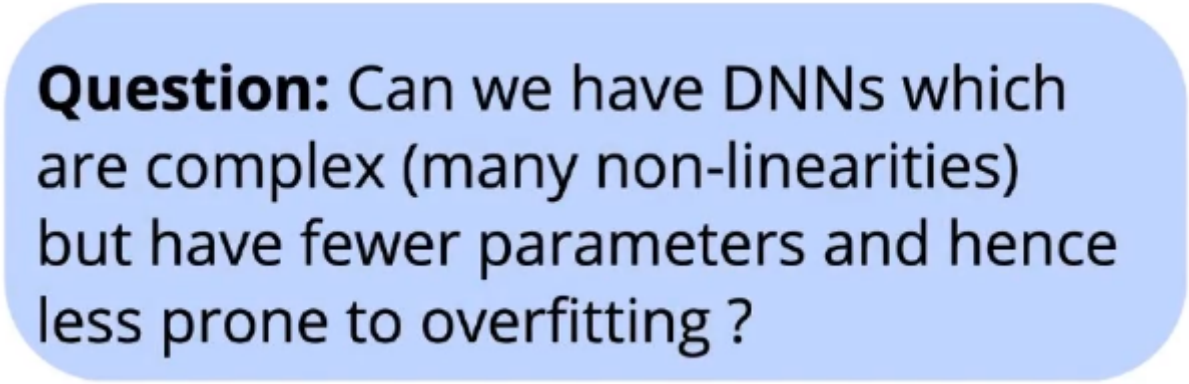


So, in this case, we have only 4 parameters for the entire image as opposed to the case of fully connected network(assuming 100 neurons in the first hidden layer, then no. of parameters in fully connected network => 16*100 = 1600).



overfitting.

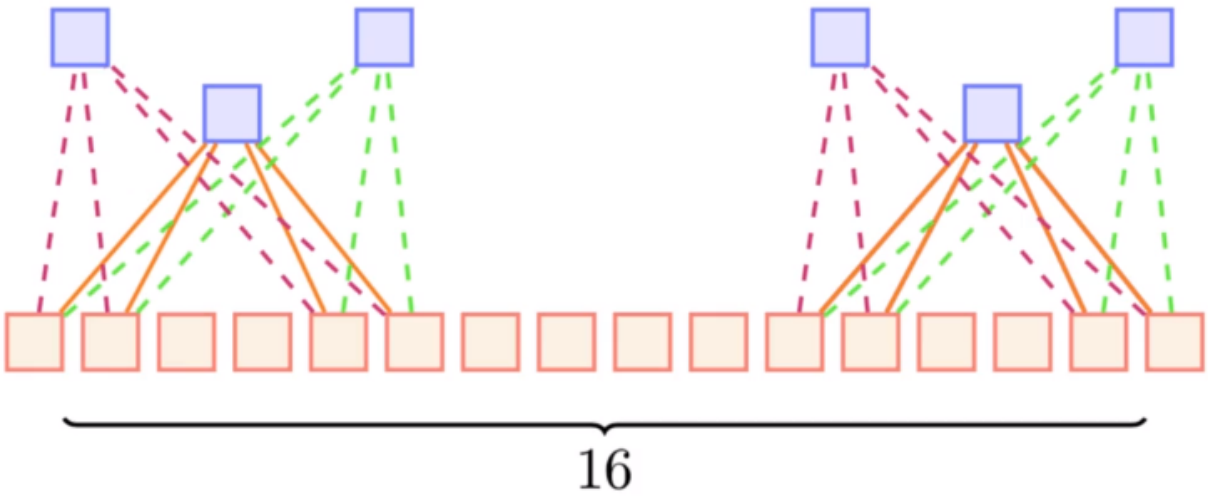
We started with the below question:

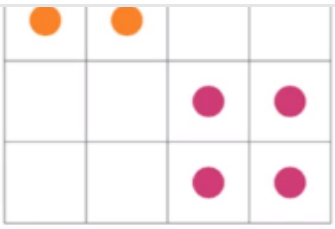


And this we have achieved in CNN's because of its weight sharing and sparse connectivity property.

Now the question under consideration is that with only 4 parameters(in the above case), would the model be able to learn the parameters and perform well on training and test set?

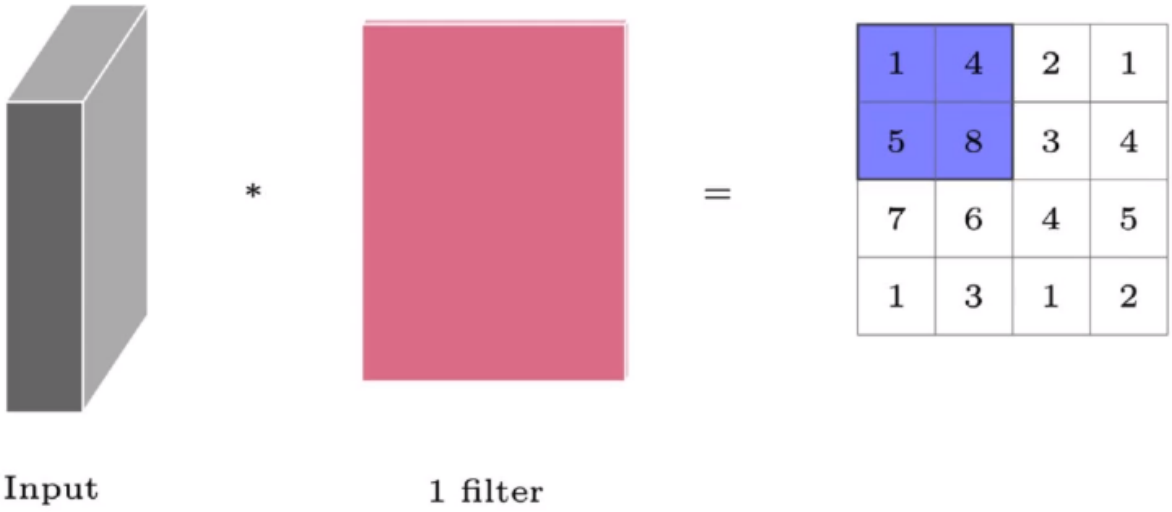
The answer to this question is that generally, we use multiple filters to capture multiple representations of the input(say 1 filter capturing edges, 1 filter for the purpose of sharpening the input and so on). And with multiple numbers of filters, let's say we use 100 filters of size 2X2, so the total number of parameters would be 100*4 = 400 which is not bad as we would have many such intermediate layers and each layer would have its own set of weights.

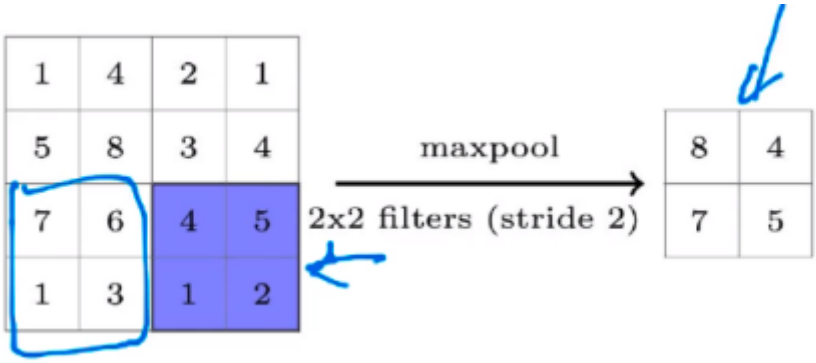




Max Pooling Operation

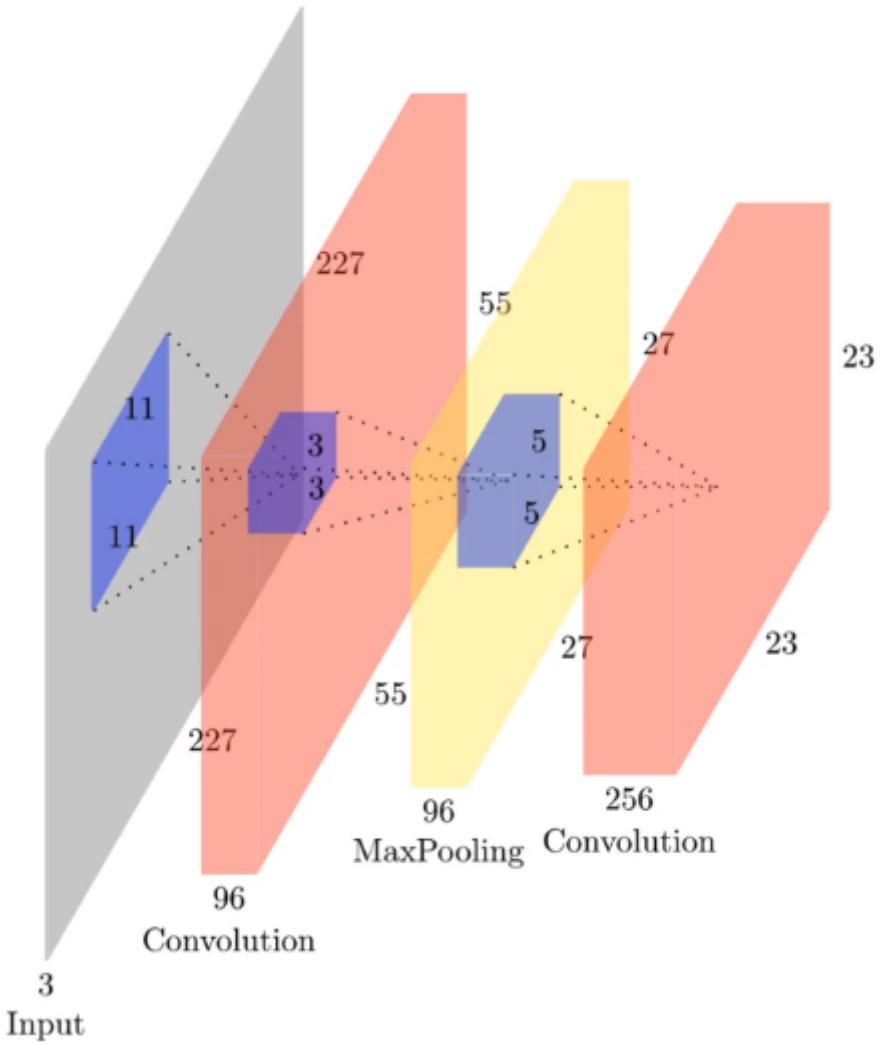
The idea here is that one feature map(output after convolving input with the filter) captures the representation for the entire image(and is typical of the same size as the input if appropriate padding is applied) and to shrink the size of the feature map, we look at a particular neighborhood and whatever be the most active part(highest number) in that neighborhood we keep that.





In the above image, max-pooling is done with **'2X2'** regions with a stride of 2 so we get the value 8 from the first part(having numbers 1, 4, 5, 8), then we move the filter by 2 in the horizontal direction and from there we get the maximum value as 4(out of 2, 1,

Max pooling is also called as sub-sampling. And sometimes instead of max pooling, average pooling is done.



We apply the non-linearity function(usually ReLU) to the output of the Convolutional operation to have non-linearity in the model and then we apply this max-pooling operation.

The intuition behind max-pooling is that say given an image, we slide across it by say **'2X2'** regions, it is just like saying from this **'2X2'** neighborhood, just tell us what is the maximum value say this is maximum blue color so we will just take it, we don't care about other finer details there, just tell us the most shining part or most elaborate part of that image and we are just taking those max. values from all the neighborhoods.

In a fully connected neural network, every neuron in each layer is connected to all neurons in the previous layers. Now assume that the input image is of size **'32 X 32 X**

be **'32 X 32 X 28 X 28 X 6'** which would be of the order of **'10^6'**.

And for this very case, if we use a CNN's with 6 filters each of size '**5 X 5 X 1**', the number of parameters required to compute the first intermediate layer would be '**6 X 5 X 5 X 1**' i.e 150. So, using only 150 parameters, we are able to go from an input of size '**32 X 32 X 1**' to a size '**28 X 28 X 6**'.

All the things like the number of filters to use, size of the filters, padding size, stride, all these are the hyper-parameters. We can experiment with these to have different network architectures.

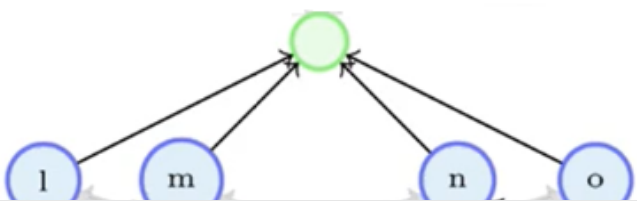
The parameters for the max pooling operation/layer is 0.

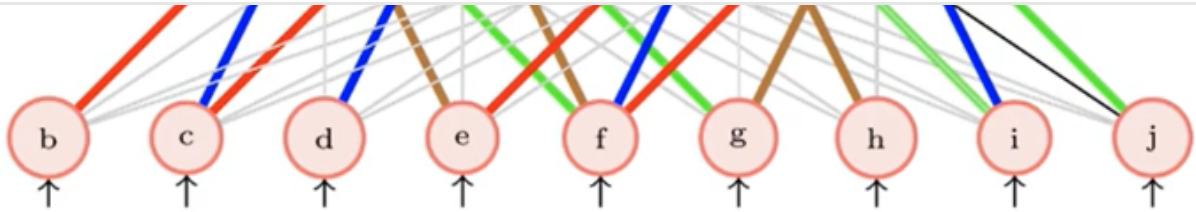
Training CNNs:

Till now, we have dealt with the Feed Forward Neural Networks(also called fully connected neural networks), where we pass the input through all the layers of the network and then we compute the final output. Based on this final output, we compute the loss value and then we take the derivative of the loss function with respect to the parameters of the model and update the parameters accordingly in a way that the overall loss is minimized.

Now CNN can be implemented as a feed-forward neural network where only a few weights are active(as at a time only a few weights are used to compute the output), we can consider the weights associated with the rest of the neurons to be 0.

So, in this case, when we backpropagate the loss function, we will only update the weights in the network which are not zero and contributed to compute the intermediate outputs as well as the final output. And this will continue throughout the training. So, we can think of it as training a fully connected neural networks where most of the weights are 0 and only a few weights participate in the computation of the output and those are the only weights that will actually get updated.





- A CNN can be implemented as a feedforward network
- wherein only a few weights (in color) are active
- the rest of the weights (in gray) are zero

Deep Learning Convolutional Network Artificial Intelligence Neural Networks

Machine Learning